

Truth versus Information in Logic Programming

Lee Naish and Harald Søndergaard

*Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
(e-mail: {lee,harald}@unimelb.edu.au)*

submitted 15 June 2012; revised 6 February 2013; accepted 20 April 2013

Abstract

The semantics of logic programs was originally described in terms of two-valued logic. Soon, however, it was realised that three-valued logic had some natural advantages, as it provides distinct values not only for truth and falsehood, but also for “undefined”. The three-valued semantics proposed by Fitting and by Kunen are closely related to what is computed by a logic program, the third truth value being associated with non-termination. A different three-valued semantics, proposed by Naish, shared much with those of Fitting and Kunen but incorporated allowances for programmer intent, the third truth value being associated with underspecification. Naish used an (apparently) novel “arrow” operator to relate the intended meaning of left and right sides of predicate definitions. In this paper we suggest that the additional truth values of Fitting/Kunen and Naish are best viewed as duals. We use Belnap’s four-valued logic, also used elsewhere by Fitting, to unify the two three-valued approaches. The truth values are arranged in a bilattice which supports the classical ordering on truth values as well as the “information ordering”. We note that the “arrow” operator of Naish (and our four-valued extension) is essentially the information ordering, whereas the classical arrow denotes the truth ordering. This allows us to shed new light on many aspects of logic programming, including program analysis, type and mode systems, declarative debugging and the relationships between specifications and programs, and successive executions states of a program. This paper is to appear in *Theory and Practice of Logic Programming (TPLP)*.

KEYWORDS: Declarative debugging, information order, intended interpretation, logic program specification, many-valued logic, modes, program analysis, specification semantics.

1 Introduction

Logic programming is an important paradigm. Computers can be seen as machines which manipulate meaningful symbols and the branch of mathematics which is most aligned with manipulating meaningful symbols is logic. This paper is part of a long line of research on what are good choices of logic to use with a “pure” subset of the Prolog programming language. We ignore the “non-logical” aspects of Prolog such as cut and built-ins which can produce side-effects, and assume a sound form

of negation (ensuring in some way that negated literals are always ground before being called).

There are several ways in which having a well-defined semantics for programs is helpful. First, it can be helpful for implementing a language (writing a compiler, for example)—it forms a specification for answering “what should this program compute”. Second, it can be helpful for writing program analysis and transformation tools. Third, it can be helpful for verification and debugging—it can allow application programmers to answer “does this program compute what I intend” and, when the answer is negative, “why not”. There is typically imprecision involved in all three cases.

1. Many languages allow some latitude to the implementor in ways that affect observable behaviour of the program, for example by not specifying the order of sub-expression evaluation (C is an example). Even in pure Prolog, typical approaches to semantics do not precisely deal with infinite loops and/or “floundering” (when a negative literal never becomes ground). Such imprecision is not necessarily a good thing, but there is often a trade-off between precision and simplicity of the semantics.
2. Program analysis tools must provide imprecise information in general if they are guaranteed to terminate, since the properties they seek to establish are almost always undecidable.
3. Programmers are often only interested in how their code behaves for some class of inputs. For other inputs they either do not know or do not care (this is in addition to the first point). Moreover, it is often convenient for programmers to reason about *partial* correctness, setting aside the issue of termination.

A primary aim of this paper is to reconcile two different uses of many-valued logic for understanding logic programs. The first use is for the provision of semantic definition, with the purpose of answering “what should this program compute?” The other use is in connection with program specification and debugging, concerned with answering “does this program compute what I intend” and similar questions involving programmer intent.

A second aim is to show the versatility of four-valued logic in a logic programming context. Four-valued logic has been recommended by Fitting (1991a), but primarily as a programming language feature, for distributed programming. In that context, the fourth truth value represents conflicting information derived from different nodes in a network. We complement that work by pointing out that motivation for four-valued logic comes from many other sources, even when we restrict attention to sequential programming. Central to our use of this logic is its support for the “information ordering” as well as the classical ordering on truth values. Our contributions are:

- We show how Belnap’s four-valued logic enables a clean distinction between a formula/query which is undefined, or non-denoting, and one which is irrelevant, or inadmissible.

- We use this logic to provide a denotational semantics for logic programs which is designed to help a programmer reason about partial correctness in a natural way. This aim is different to the semanticist’s traditional objective of reflecting runtime behaviour, or aligning denotational and operational semantics.
- The approximative nature of logic program analysis naturally fits with the information ordering and we show how semantic approximation can be expressed in terms of four truth values.
- We show how four-valued logic helps modelling the concept of modes in a moded logic programming language such as Mercury.
- We argue that a four-valued semantics and the information ordering clarify the relation between programs and formal specifications.
- We show how established practice in declarative debugging can be extended with four values.
- Finally, we argue that the computation model of logic programming can be viewed from the perspective of the information order rather than the classical truth order.

This paper is an extended version of Naish et al. (2012). We assume the reader has a basic understanding of pure logic programs, including programs in which clause bodies use negation, and their semantics. We also assume the reader has some familiarity with the concepts of types and modes as they are used in logic programming.

The paper is structured as follows. We set the scene in Section 2 by revisiting the problems that surround approaches to logical semantics for pure Prolog. In Section 3 we introduce the three- and four-valued logics and many-valued interpretations that the rest of the paper builds upon. In Section 4 we provide some background on different approaches to the semantics of pure Prolog, focusing on work by Fitting and Kunen. In Section 5 we review Naish’s approach to what we call specification semantics. In Section 6 we present a new four-valued approach which combines two three-valued approaches (Fitting and Naish). Section 7 establishes a property of this semantics analogous to model intersection. Section 8 shows how four-valued logic naturally captures the kind of approximation employed in program analysis. Section 9 shows how it also helps with modelling the concept of *modes* in a moded logic language such as Mercury. Section 10 discusses its relevance for formal specification and Section 11 sketches its application to declarative debugging. Section 12 shows how the logic programming computation model can be seen in terms of the information ordering. Section 13 discusses some additional related work and Section 14 concludes.

2 Logic programs

Suppose we need Prolog predicates to capture the workings of classical propositional disjunction and negation. We may specify the behaviour exhaustively (we use `neg` for negation since `not` is often used as a general negation primitive in Prolog):

<code>or(t, t, t).</code>	<code>neg(t, f).</code>
<code>or(t, f, t).</code>	<code>neg(f, t).</code>
<code>or(f, t, t).</code>	
<code>or(f, f, f).</code>	

yielding simple, correct predicates. If we also need a predicate for implication, we could define

`implies(X, Y) :- neg(X, U), or(U, Y, t).`

Clauses are universally closed. Stated differently, the variables in the head of a clause are universally quantified over the whole clause; those which only occur in the body are existentially quantified within the body.

Although Prolog programs explicitly define only what is true, it is also important that they implicitly define what is false. This is the case for most programs and is essential when negation is used. For example, `neg(t, t)` would be considered false and for it to succeed would be an error. Because (implicit) falsehood depends on the set of all clauses defining a predicate, it is often convenient to group all clauses into a single definition with distinct variables in the arguments of the clause head. This can be done in Prolog by using the equality (=) and disjunction (;) primitives. For example, `neg` could be defined

`neg(X, Y) :- (X=t, Y=f ; X=f, Y=t).`

Clark (1978) defined the *completion* of a logic program which explicitly groups clauses together in this way; others (Fitting 1991a; Naish 2006) assume the program contains a single clause per predicate from the outset. Henceforth we assume the same. The `:-` in a single-clause definition thus tells us about both the truth and falsehood of instances of the head. Exactly how `:-` is best viewed has been the topic of much debate and is a central focus of this paper. One issue is the relationship between the truth values of the head and body—what set of truth values do we use, what constitutes a model or a fixed point, etc. Another is whether we consider one particular model/fixed point (such as the least one according to some ordering) as the semantics or do we consider any one of them to be a possible semantics or consider the set of all models/fixed points as the semantics.

Let us fix our vocabulary for logic programs and lay down an abstract syntactic form.

Definition 1 (Syntax)

An atom (or atomic formula) is of the form $p(t_1, \dots, t_n)$, where p is a predicate symbol (of arity n) and t_1, \dots, t_n are terms. If $A = p(t_1, \dots, t_n)$ then A 's predicate symbol $\text{pred}(A)$ is p . There is a distinguished equality predicate $=$ with arity 2, written using infix notation. A *literal* is an *atom* A or the negation of an atom, written $\neg A$. A *conjunction* C is a conjunction of literals. A *disjunction* D is of the form $C_1 \vee \dots \vee C_k$, $k > 0$, where each C_i is a conjunction. For a syntactic object o (literal, clause, disjunction, and so on), we use $\text{vars}(o)$ to denote the set of variables that occur in o .

A *predicate definition* is a pair $(H, \exists W[D])$ where H is an atom in most general

form $p(V_1, \dots, V_n)$ (that is, the V_i are distinct variables), D is a disjunction, and $W = \text{vars}(D) \setminus \text{vars}(H)$. We call H the *head* of the definition and $\exists W[D]$ its *body*. The variables $\text{vars}(H)$ are the *head variables* and the variables W are *local variables*. Finally, a *program* is a finite set S of predicate definitions such that if $(H_1, B_1) \in S$ and $(H_2, B_2) \in S$ then $\text{pred}(H_1) \neq \text{pred}(H_2)$.

In program text we use Prolog notation and assume this is converted to the abstract syntax described above by combining clauses and mapping “,”, “;” and “**not**” to \wedge , \vee and \neg , respectively, etc. For example, the definition of **implies/3** above is shorthand for $(\text{implies}(X, Y), \exists U[\text{neg}(X, U) \wedge \text{or}(U, Y, t)])$.

We let \mathcal{G} denote the set of ground (that is, variable-free) atoms (for some suitably large fixed alphabet).

Definition 2 (Head instance, head grounding)

A *head instance* of a predicate definition $(H, \exists W[D])$ is an instance where all head variables have been replaced by other terms, and all local variables remain unchanged. A *head grounding* is a head instance where the head is ground.

For example, $(\text{implies}(t, f), \exists U[\text{neg}(t, U) \wedge \text{or}(U, f, t)])$ is a head grounding of the **implies/3** definition. Later we shall define models and “immediate consequence” functions for two-, three-, and four-valued semantics. The use of *head groundings*, rather than more conventional approaches is a technical convenience which allows us to emphasize the relationship between models and immediate consequence.

3 Interpretations and models

In two-valued logic, an interpretation is a mapping from \mathcal{G} to $\mathbf{2} = \{\mathbf{f}, \mathbf{t}\}$. To give meaning to recursively defined predicates, the usual approach is to impose some structure on $\mathcal{G} \rightarrow \mathbf{2}$, to ensure that we are dealing with a lattice, or a semi-lattice at least. Given the traditional “closed-world” assumption (that a formula is false unless it can be proven true), the natural ordering on $\mathbf{2}$ is this: $b_1 \leq b_2$ iff $b_1 = \mathbf{f} \vee b_2 = \mathbf{t}$. The ordering on interpretations is the natural (pointwise) extension of \leq , equipped with which $\mathcal{G} \rightarrow \mathbf{2}$ is a complete lattice.

Three-valued logic is arguably a more natural logic for the partial predicates that emerge from pure Prolog programs, and more generally, for the partial functions that emerge from programming in any Turing complete language. The case for three-valued logic as the appropriate logic for computation has been made repeatedly, starting with Kleene (1938) and pursued by the VDM school (for example Barringer et al. (1984), Jones and Middelburg (1994)), and others. The third value, **u**, for “undefined”, finds natural uses, for example as the value of $\mathbf{p}(\mathbf{b})$, given the program in Figure 1.

With three- or four-valued logic, an interpretation becomes a mapping from \mathcal{G} to $\mathbf{3} = \{\mathbf{u}, \mathbf{f}, \mathbf{t}\}$ or to $\mathbf{4} = \{\mathbf{u}, \mathbf{f}, \mathbf{t}, \mathbf{i}\}$ (we discuss the role of the fourth value **i** shortly). For compatibility with the way equality is treated in Prolog, we constrain interpretations so $x = y$ is mapped to **t** if x and y are identical (ground) terms, and to **f** otherwise. This is irrespective of the set of truth values used. There are different

```

p(a) .
p(b) :- p(b) .
p(c) :- not p(c) .
p(d) :- not p(a) .

```

Fig. 1. Small program to exemplify semantics

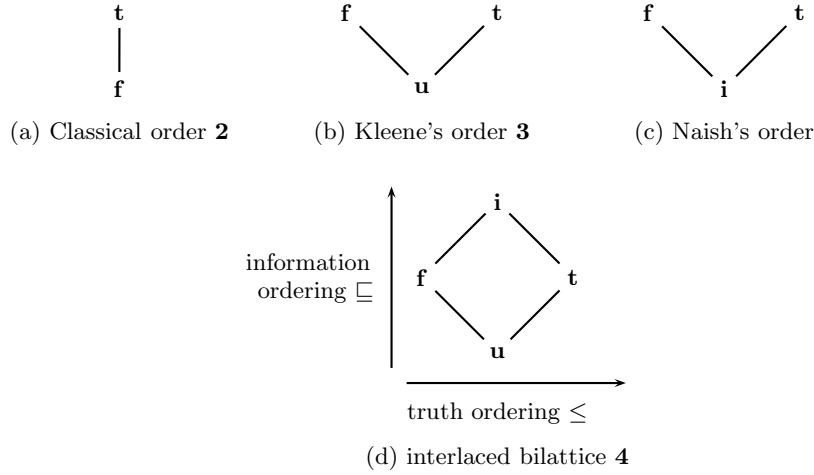


Fig. 2. Partially ordered sets of truth values

choices for the semantics of the connectives. In Section 4.3 we discuss connectives and give particular truth tables for the common connectives, corresponding to Belnap's four-valued logic (Belnap 1977) (the restriction to three-valued logic that results from deleting rows and columns containing **i** corresponds to Kleene's (strong) three-valued logic K_3 (Kleene 1938)).

We denote the ordering depicted in Figure 2(b) by \sqsubseteq ,¹ that is, $b_1 \sqsubseteq b_2$ iff $b_1 = \mathbf{u} \vee b_1 = b_2$, and we overload this symbol to also denote the ordering in Figure 2(d) (that is, $b_1 \sqsubseteq b_2$ iff $b_1 = \mathbf{u} \vee b_1 = b_2 \vee b_2 = \mathbf{i}$), as well of the natural extensions to $\mathcal{G} \rightarrow \mathbf{3}$ or $\mathcal{G} \rightarrow \mathbf{4}$. We shall also use \sqsupseteq , the inverse of \sqsubseteq . In some contexts we disambiguate the symbol by using a superscript: \sqsubseteq^3 or \sqsubseteq^4 . Similarly, we use \geq^2 for the truth ordering with two values, and $=^2$, $=^3$ and $=^4$ for equality of truth values in the different domains. When the context allows, we write the partially ordered set $(\mathbf{2}, \leq)$ simply as **2**, $(\mathbf{3}, \sqsubseteq^3)$ as **3**, and $(\mathbf{4}, \sqsubseteq^4)$ as **4**.

The structure in Figure 2(d) is the simplest of Ginsberg's so-called bilattices (Ginsberg 1988). The diamond shape can be considered a lattice from two distinct angles. The ordering \leq is the “truth” ordering, whereas \sqsubseteq is the “information” ordering. For the truth ordering we denote the meet and join operations by \wedge and \vee , respectively. For the information ordering we denote the meet and join operations by \sqcap and \sqcup , respectively. Thinking of the four elements as *sets* of classical values,

¹ While (b) and (c) are structurally identical, **u** and **i** carry different meanings, as discussed later.

with $\mathbf{u} = \emptyset$, $\mathbf{i} = \{\mathbf{f}, \mathbf{t}\}$, and \mathbf{f} and \mathbf{t} being singleton sets, the information ordering is simply the subset ordering. Regarding the truth ordering, note that $b_1 \leq b_2$ holds if and only if b_2 is at least as true as b_1 , and at the same time no more false. That is, we can move up in the truth value ordering by adding truth, or removing falsehood, or both. The bilattice in Figure 2(d) is interlaced: Each meet and each join operation is monotone with respect to *either* ordering. The bilattice is also distributive in the strong sense that each meet and each join operation distributes over all the other meet and join operations.

An equivalent view of three- or four-valued interpretations is to consider an interpretation to be a pair of ground atom sets. That is, the set of interpretations $\mathcal{I} = \mathcal{P}(\mathcal{G}) \times \mathcal{P}(\mathcal{G})$. In this view an interpretation $I = (T_I, F_I)$ is a set T_I of ground atoms deemed true together with a set F_I of ground atoms deemed false. A ground atom A that appears in neither is deemed undefined. Such a truth value *gap* may arise from the absence of any evidence that A should be true, or that A should be false. In a four-valued setting, para-consistency is a possibility: A ground atom A may belong to $T_I \cap F_I$. Such a truth value *glut* may arise from the presence of conflicting evidence regarding A 's truth value.

The concept of a *model* is central to many approaches to logic programming. A model is an interpretation which satisfies a particular relationship between the truth values of the head and body of each head grounding. We now define how truth for atoms is lifted to truth for bodies of definitions.

Definition 3 (Made true)

Let $I = (T_I, F_I)$ be an interpretation. Recall that a ground equality atom is in T_I or F_I , depending on whether its arguments are one and the same term.

For a ground atom A ,

I makes A true iff $A \in T_I$
 I makes A false iff $A \in F_I$

For a ground negated atom $\neg A$,

I makes $\neg A$ true iff $A \in F_I$
 I makes $\neg A$ false iff $A \in T_I$

For a ground conjunction $C = L_1 \wedge \dots \wedge L_n$,

I makes C true iff $\forall i \in \{1, \dots, n\}$ I makes L_i true
 I makes C false iff $\exists i \in \{1, \dots, n\}$ I makes L_i false

For a ground disjunction $D = C_1 \vee \dots \vee C_n$,

I makes D true iff $\exists i \in \{1, \dots, n\}$ I makes C_i true
 I makes D false iff $\forall i \in \{1, \dots, n\}$ I makes C_i false

For the existential closure of a disjunction $\exists W[D]$,

I makes $\exists W[D]$ true iff
 I makes some ground instance of D true
 I makes $\exists W[D]$ false iff
 I makes all ground instances of D false

We use this to extend interpretations naturally so they map \mathcal{G} and existential closures of disjunctions to **2**, **3** or **4**. We freely switch between viewing an interpretation as a mapping and as a pair of sets. Thus, for any formula F ,

$$I(F) = \begin{cases} \mathbf{u} & \text{if } I \text{ neither makes } F \text{ true nor false} \\ \mathbf{f} & \text{if } I \text{ makes } F \text{ false and not true} \\ \mathbf{t} & \text{if } I \text{ makes } F \text{ true and not false} \\ \mathbf{i} & \text{if } I \text{ makes } F \text{ true and also false} \end{cases}$$

Definition 4 ($\mathcal{R}^{\mathcal{D}}$ -Model)

Let \mathcal{D} be **2**, **3** or **4** and $\mathcal{R}^{\mathcal{D}}$ be a binary relation on \mathcal{D} . An interpretation I is an $\mathcal{R}^{\mathcal{D}}$ -model of predicate definition (H, B) if, for each head grounding $(H\theta, B\theta)$, we have $\mathcal{R}^{\mathcal{D}}(I(H\theta), I(B\theta))$. I is an $\mathcal{R}^{\mathcal{D}}$ -model of program P if it is an $\mathcal{R}^{\mathcal{D}}$ -model of every predicate definition in P .

For example, a $=^2$ -model is a two-valued interpretation where the head and body of each head grounding have the same truth value.

Another important concept used in logic programming semantics and analysis is the “immediate consequence operator”. The original version, T_P , took a set of true atoms (representing a two-valued interpretation) and returned the set of atoms which could be proved from those atoms by using some clause of program P for a single deduction step.² Here we give an equivalent definition based on how we define interpretations. We write Φ_P for the immediate consequence operator, following Fitting (1985). Note, however, that we give Φ_P a definition in terms of *head groundings* (Definition 2), and the same definition is used for the two-, three-, and four-valued cases alike.

Definition 5 (Φ_P)

Given an interpretation I and program P , $\Phi_P(I)$ is the interpretation I' such that the truth value of an atom H in I' is the truth value of B in I , where (H, B) is a head grounding of a definition in P .

Proposition 1

Let \mathcal{D} be **2**, **3** or **4**. A (\mathcal{D} -) interpretation I is a fixed point of Φ_P iff I is a $=^{\mathcal{D}}$ -model of P .

Proof

This follows easily from the given definitions. Assume $\Phi_P(I) = I$. Then, by definition of $\Phi_P(I)$, for each head grounding (H, B) of some predicate definition in P , $I(H) =^{\mathcal{D}} I(B)$. That is, I is a $=^{\mathcal{D}}$ -model of P . Conversely, assume I is a $=^{\mathcal{D}}$ -model of each predicate definition (H, B) . That is, $H\theta =^{\mathcal{D}} B\theta$ for all θ . Then, by definition of Φ_P , $\Phi_P(I) = I$. \square

² The original version, due to van Emden and Kowalski (1976), used ‘ T ’, but T_P has become standard. Various definitions which generalise T_P to **3** and **4** have been given (Apt and Bol 1994).

4 Logic program operational semantics

We first discuss some basic notions and how Clark’s two-valued approach to logic program semantics fits with what we have presented so far. Then we discuss the Fitting/Kunen three-valued approach and Fitting’s four-valued semantics.

4.1 Two-valued semantics

There are three aspects to the semantics of logic programs: proof theory, model theory and fixed point theory (see Lloyd (1984), for example). The proof theory is generally based on resolution, often some variant of SLDNF resolution (Clark 1978). This gives a top-down operational semantics, which is not our main focus but is briefly discussed in Section 12. The model theory gives a declarative view of programs and is particularly useful for high level reasoning about partial correctness. The fixed point semantics, based on Φ_P or T_P , gives an alternative “bottom up” operational semantics (which has been used in deductive databases) and which is also particularly useful for program analysis.

The simplest semantics for pure Prolog disallows negation and treats a Prolog program as a set of definite clauses. Prolog’s $:-$ is treated as classical implication, \leftarrow , that is, \geq^2 -models are used. There is an important soundness result: if the programmer has an intended interpretation which is a model, any ground atom which succeeds is true in that model. The (\leq) least model is also the least $=^2$ -model and the least fixed point of Φ_P , which is monotone in the truth ordering (so a least fixed point always exists). The set of true atoms in this least model is the set of atoms which are true in all \geq^2 -models (and $=^2$ -models) and is also the set of atoms which have a successful derivation using SLD resolution. For these reasons, this is the accepted semantics for Prolog programs without negation.

To support negation in the semantics, Clark (1978) combined all clauses defining a particular predicate into a single “if and only if” definition which uses the classical bi-implication \leftrightarrow . This is called the Clark completion $comp(P)$ of a program P . Our definitions are essentially the same, but we avoid the \leftrightarrow symbol. In this paper’s terminology, Clark used $=^2$ -models, which correspond to classical fixed points of Φ_P . Clark specifically considered logical consequences of $comp(P)$: atoms which were true in all $=^2$ -models.

The soundness result above applies, and any finitely failed ground atom must also be false in the programmer’s intended interpretation, if this interpretation is a model. However, because Φ_P is non-monotone in the truth ordering when negation is present, there may be multiple minimal fixed points/models, or there may be none. For example, using Clark’s semantics for the program in Figure 1, there is no model and no fixed point due to the clause for $p(c)$, yet the query $p(a)$ succeeds and $p(d)$ finitely fails. Thus the Clark semantics does not align particularly well with the operational semantics.

4.2 Three-valued semantics

Even in the absence of negation, a two-valued semantics is lacking in its inability to distinguish failure and looping. Mycroft (1984) explored the use of many-valued logics, including **3**, to remedy this. Mycroft discussed this for Horn clause programs, and others, including Fitting (1985) and Kunen (1987), subsequently adapted Clark's work to a three-valued logic, addressing the problem of how to account properly for the use of explicit negation in programs.

In a two-valued setting, the Clark completion may be inconsistent, witness the completion of the clause for $p(c)$ in Figure 1. Hence the Clark completion is unable to give a reasonable meaning to $p(a)$, $p(b)$, and $p(d)$, even though these atoms do not depend on $p(c)$. If we were to delete the clause for $p(c)$ in Figure 1, the Clark semantics would map $p(b)$ to **f**, even though it does not finitely fail. The reason is that the smallest 2-valued model of the Clark completion $p(b) \Leftrightarrow p(b)$ maps $p(b)$ to **f**.

However, a $=^3$ -model always exists for a Clark-completed program; for example, $p(c)$ takes on the third truth value. Moreover, since Φ_P is monotone with respect to the information ordering, a least fixed point always exists and coincides with the least $=^3$ -model. Ground atoms which are **t** in this model (such as $p(a)$ in Figure 1) are those which have successful derivations, while ground atoms which are **f** (such as $p(d)$) are those which have finitely failed SLDNF trees (Clark 1978). Atoms with the third truth value ($p(b)$ and $p(c)$) must loop. Atoms which are **t** or **f** in the Fitting/Kunen semantics may also loop if the search strategy or computation rule are unfair (even without negation, **t** atoms may loop with an unfair search strategy). Furthermore, when negation is present, a computation may *flounder* owing to a negated call which never becomes ground and hence is never selected (this is a fourth possible behaviour). However the Fitting/Kunen approach does align the model theoretic and fixed point semantics much more closely to the operational semantics of Prolog than the approach of Clark, and we can imagine an idealised logic programming language where the alignment is precise.

Φ_P has a drawback, though: while monotone, it is not in general continuous. Blair (1982) shows that the smallest ordinal β for which $\Phi_P^\beta(\perp)$ is the least fixed point of Φ_P may not be recursive³ Kunen (1987) shows that, with a semantics based on three-valued Herbrand models (all models or the least model), the set of ground atoms true in such models may not be recursively enumerable⁴. Kunen instead suggests a semantics based on any three-valued model and shows that truth (**t**) in all $=^3$ -models is equivalent to being deemed true by $\Phi_P^n(\perp)$ for some $n \in \mathbb{N}$. Hence Kunen proposes $\Phi_P^\omega(\perp)$ as the meaning of program P . For a given P and ground atom A , it is decidable whether A is **t** in $\Phi_P^n(\perp)$, so whether A is **t** in $\Phi_P^\omega(\perp)$ is semi-decidable.

For simplicity, in this paper we take (the possibly non-computable) $M = lfp(\Phi_P)$

³ The (possibly transfinite) powers of Φ_P are defined the standard way: For a successor ordinal β , $\Phi_P^\beta(x) = \Phi_P(\Phi_P^{\beta-1}(x))$, and for a limit ordinal β , $\Phi_P^\beta(x) = \bigsqcup_{\alpha < \beta} \Phi_P^\alpha(x)$.

⁴ We use \perp to denote the smallest interpretation with respect to \sqsubseteq .

to be the meaning of a program, that is, the least $=^3$ -model. However, since we shall be concerned with over-approximations to M , what we shall have to say will apply equally well if Kunen's $\Phi_P^\omega(\perp)$ is assumed.

4.3 Four-valued semantics

Subsequent to his three-valued proposal, Fitting recommended, in a series of papers (1988; 1989; 1991b; 1991a; 2002), bilattices as suitable bases for logic program semantics. The bilattice **4** (Figure 2(d)) was just one of several studied for the purpose, and arguably the most important one.

Fitting's motivation for employing four-valued logic was, apart from the elegance of the interlaced bilattices and their algebraic properties, the application in a logic programming language which supports a notion of (spatially) distributed programs. In this setting there is a natural need for a fourth truth value, \top (our **i**), to denote conflicting information received from different nodes in a computing network.

In the language proposed by Fitting (1991a), the traditional logical connectives used on the right-hand sides of predicate definitions are explained in terms of the truth ordering. Negation is reflection in the truth ordering: $\neg \mathbf{u} = \mathbf{u}$, $\neg \mathbf{f} = \mathbf{t}$, $\neg \mathbf{t} = \mathbf{f}$ and $\neg \mathbf{i} = \mathbf{i}$, conjunction is meet (\wedge), disjunction is join (\vee), and existential quantification is the least upper bound (\bigvee) of all instances. These tables give conjunction, disjunction and negation in **4**:

\wedge	u	t	f	i
u	u	u	f	f
t	u	t	f	i
f	f	f	f	f
i	f	i	f	i

\vee	u	t	f	i
u	u	t	u	t
t	t	t	t	t
f	u	t	f	i
i	t	t	i	i

\neg	u	t	f	i
u	u	t	f	i
t	u	t	f	i
f	u	t	f	i
i	u	t	f	i

The operations \sqcap and \sqcup are similarly given by Figure 2(d). Fitting refers to \sqcap (he writes \otimes) as *consensus*, since $x \sqcap y$ represents what x and y agree about. The \sqcup operation (which he writes as \oplus) he refers to as *gullibility*, since $x \sqcup y$ represents agreement with both x and y , whatever they say, including cases where they disagree. Palmer (1997) also uses this logic with another parallel logic programming language, Andorra Kernel Language. Although AKL does not support \sqcap or \sqcup as explicit language primitives, Palmer's AKL compiler uses such operations in its analysis of parallel sub-computations which may or may not agree on their results.

The idea of an information (or knowledge) ordering is familiar to anybody who has used domain theory and denotational semantics. To give meaning to recursively defined objects we refer to fixed points of functions defined on structures equipped with some ordering—the information ordering. This happens already in the three-valued approaches to semantics discussed above. Three-valued semantics does use the distinction between a truth ordering \leq and an information ordering \sqsubseteq , but it does not expose it as radically as the bilattice. In Fitting's words, the three-valued approach, “while abstracting away some of the details of [Kripke's theory of truth] still hides the double ordering structure” (Fitting 2006).

The logic programming language of Fitting (1991a) contains operators \otimes and \oplus , reflecting the motivation in terms of distributed programs. We, on the other hand, deal with a language with traditional pure Prolog syntax. If the task was simply to model its operational semantics, having four truth values rather than three would offer little, if any, advantage. However, our motivation for using four-valued logic is very different to that of Fitting. We find compelling reasons for the use of four-valued logic to explain certain programming language features, as well as to embrace, semantically, such software engineering aspects as program correctness with respect to programmer intent or specification, declarative debugging, and program analysis. We next discuss one of these aspects.

5 Three-valued specification semantics

Naish (2006) proposed an alternative three-valued semantics. Unlike other approaches, the objective was not to align declarative and operational semantics. Instead, the aim was to provide a declarative semantics which can help programmers develop correct code in a natural way. Naish argued that intentions of programmers are not two-valued. It is generally intended that some ground atoms should succeed (be considered **t**) and some should finitely fail (be considered **f**) but some should never occur in practice; there is no particular intention for how they should behave and the programmer does not care and often does not know how they behave. An example is merging lists, where it is assumed two sorted lists are given as input: it may be more appropriate to consider the value of `merge([3,2],[1],[1,3,2])` *irrelevant* than to give it a classical truth value, since a *precondition* is violated. Or consider this program:

<pre>or2(t, _, t). or2(f, B, B).</pre>	<pre>or3(_, t, t). or3(B, f, B).</pre>
--	--

It gives two alternative definitions of `or` (previously defined in Section 2), both designed with the assumption that the first two arguments will always be Booleans. If they are not, we consider the atom to be *inadmissible* (a term used in debugging (Pereira 1986; Naish 2000)) and give it the truth value **i**. Interpretations can be thought of as the programmer's understanding of a specification, where **i** is used for underspecification of behaviour. The same three-valued interpretation can then be used with all three definitions of `or`. A programmer can first fix the interpretation then code any of these definitions and reason about their correctness. In contrast, both the Clark and Fitting/Kunen semantics assign different, incompatible meanings to the three definitions, with atoms such as `or3(4,f,4)` and `or2(t,[],t)` considered **t** and `or3(t,[],t)` considered **f**. In order for the programmer's intended interpretation to be a $=^2$ -model or $=^3$ -model, unnatural distinctions such as these must be made. Naish (2006) argues that it is unrealistic for programmers to use such interpretations as a basis for reasoning about correctness of their programs. In Section 6 we consider a somewhat larger example in more depth.

Although Naish uses **i** instead of **u** as the third truth value, his approach is structurally the same as Fitting/Kunen's with respect to the ordering, Figure 2(b)

and (c), the Φ_P operator and the meaning of connectives used in the body of definitions. The key technical difference is how Prolog’s $:-$ is interpreted. Fitting generalises Clark’s classical \leftrightarrow to \cong or “strong equivalence”, where heads and bodies of head groundings must have the same truth values. Naish defined a different “arrow”, \leftarrow , which is asymmetric, but not a conservative extension of classical implication (so the choice of symbol is perhaps misleading). In addition to identical truth values for heads and bodies, Naish allows head groundings of the form (\mathbf{i}, \mathbf{f}) and (\mathbf{i}, \mathbf{t}) . The difference is captured by these tables (Fitting left, Naish right):⁵

\cong	\mathbf{t}	\mathbf{f}	\mathbf{u}
\mathbf{t}	\mathbf{t}	\mathbf{f}	\mathbf{f}
\mathbf{f}	\mathbf{f}	\mathbf{t}	\mathbf{f}
\mathbf{u}	\mathbf{f}	\mathbf{f}	\mathbf{t}

\leftarrow	\mathbf{t}	\mathbf{f}	\mathbf{i}
\mathbf{t}	\mathbf{t}	\mathbf{f}	\mathbf{f}
\mathbf{f}	\mathbf{f}	\mathbf{t}	\mathbf{f}
\mathbf{i}	\mathbf{t}	\mathbf{t}	\mathbf{t}

Naish’s arrow captures the principle that, if a predicate is called in an inadmissible way, then it does not matter if it succeeds or fails. The definition of a model uses this weaker “arrow”; we discuss it further in Section 6. Naish (2006) shows that for any model, only \mathbf{t} and \mathbf{i} atoms can succeed and only \mathbf{f} and \mathbf{i} atoms can finitely fail. In models of the code in Figure 1, $\mathbf{p}(\mathbf{b})$ can be \mathbf{t} or \mathbf{f} or \mathbf{i} but $\mathbf{p}(\mathbf{c})$ can only be \mathbf{i} . For practical code, programmers can reason about partial correctness using intuitive models in which the behaviour of some atoms is unspecified.

6 Four-valued specification semantics

The Fitting/Kunen and Naish approaches all use three truth values, the Kleene strong three-valued logic for the connectives in the body of definitions, and the same immediate consequence operator. It is thus tempting to assume that the “third” truth value in these approaches is the same in some sense. This is implicitly assumed by Naish (2006) when he compares different approaches. However, the third truth value is used for very different purposes in the approaches being compared. Fitting (1985) and Kunen (1987) use it to make the semantics more precise than Clark—distinguishing success and finite failure from nontermination (neither success nor finite failure). Naish (2006) uses it to make the semantics *less* precise than Clark, allowing a truth value corresponding to success or finite failure. Thus we believe it is best to treat the third truth values of Fitting/Kunen and Naish as *duals* instead of the same value. Naish treats \mathbf{i} as the bottom element whereas in **4** it is more naturally the top element, with the ordering of Naish inverted. Because conjunction, disjunction and negation in **4** are symmetric in the information order, the third value in the Kleene strong three-valued logic can map to either the top or bottom element in **4**. This is why the third truth values in Fitting/Kunen and Naish are treated identically, even though they are better viewed as semantically distinct.

⁵ We abuse notation here: \cong and \leftarrow are not actually used as connectives, so the table entries should really be “model” and “not model” rather than \mathbf{t} and \mathbf{f} .

```

% Checks A-B = E-F, where all are natural numbers,
% represented in Peano style with 0 and s/1
% This definition is common to programs P1-P4
eq_diff(A, B, E, F) :- sub(A, B, D), sub(E, F, D).

% sub/3 definition for P1
sub(0, 0, 0).
sub(s(A), 0, s(D)) :- sub(A, 0, D).
sub(s(A), s(B), D) :- sub(A, B, D).

% sub/3 definition for P3
sub(A, A, 0).
sub(A, B, s(D)) :-
    not(A=B), sub(A, s(B), D).

% sub/3 definition for P2
sub(A, 0, A).
sub(s(A), s(B), D) :- sub(A, B, D).

% sub/3 definition for P4
sub(A, A, 0).
sub(A, B, s(D)) :- sub(A, s(B), D).

```

Fig. 3. Programs P1–P4 for subtraction over natural numbers

The four values **t**, **f**, **i** and **u** are associated with truth/success, falsehood/finite failure, inadmissibility (the Naish third value) and looping/error (the Fitting/Kunen third value). Inadmissibility can be seen as saying that both success and failure are correct, so we can see it as the union of both. Atoms which are **u** in the Fitting/Kunen semantics neither succeed nor finitely fail. Thus, as already pointed out, the information ordering can also be seen as the set ordering, \subseteq , if we interpret the truth values in **4** as sets of Boolean values.

Consider the four programs depicted in Figure 3, which have a common definition of `eq_diff/4` (which checks if the differences of two pairs of natural numbers are the same) but different definitions of `sub/3` (which performs subtraction over natural numbers). These programs have different sets of ground atoms which succeed and finitely fail; we discuss these in more detail later. For some of the programs there are ground atoms which neither succeed nor finitely fail and the Fitting/Kunen semantics has the advantage of reflecting this whereas Clark's cannot. For example, the same set of atoms succeed in P3 and P4 (the least two-valued models, used by Clark, exist and are the same) but some atoms such as `eq_diff(s(0), 0, 0, s(0))` finitely fail in P3 but loop in P4 (so the least three-valued models differ). Naish's semantics has the advantage of allowing the programmer to reason about correctness with respect to intentions or specifications which are imprecise. For example, a programmer may only want to specify the desired behaviour of ground atoms `eq_diff(A,B,E,F)` where all arguments are natural numbers (of the form $s^n(0)$) and $A-B$ and $E-F$ are defined over natural numbers; other atoms can reasonably be considered inadmissible. This allows us to simply establish partial correctness of all four programs — the behaviours differ, but only for atoms the programmer does not care about. The four-valued semantics which we propose here combines the advantages of the Fitting/Kunen and Naish approaches within a single unified framework.

We now show how Naish's semantics can be combined with that of Fitting/Kunen and generalised to **4**. Adding the truth value **i** is a conservative extension to the Fitting/Kunen semantics. The three-valued fixed points of Φ_P are preserved, including the least fixed point, and the way the semantics describes what is *computed*

is unchanged, though the additional truth value can be useful for *approximating* what is computed. However, adding the truth value **u** to the Naish semantics *does* allow us to describe more precisely what is *intended*. There are occasions when both the success and finite failure of an atom are considered incorrect behaviour and thus **u** is an appropriate value to use in the intended interpretation. We give three examples. The first is an interpreter for a Turing-complete language. If the interpreter is given (the representation of) a looping program it should not succeed and it should not fail. The second is an operating system. Ignoring the details of how interaction with the real world is modelled in the language, termination means the operating system crashes. The third is code which is only intended to be called in limited ways, but is expected to be robust and check its inputs are well formed. Exceptions or abnormal termination with an error message are best not considered success or finite failure. Treating them in the same way as infinite loops in the semantics may not be ideal but it is more expressive than using the other three truth values (indeed, “infinite” loops are never really infinite because resources are finite and hence some form of abnormal termination results).

Naish (2006) defines models in terms of the \leftarrow described earlier and shows that I is a model if and only if $I \sqsupseteq^3 \Phi_P(I)$. The significance of this proposition is not noted by Naish (2006), but it prompts a key observation: the \leftarrow defines the information order on truth values! The classical arrow defines the truth ordering on two values; Naish’s arrow defines the orthogonal ordering in the three-valued extension. It is therefore clear how Naish’s arrow can be generalised to **4**. The models of Naish (2006) are \sqsupseteq^3 -models, which can be generalised to \sqsupseteq^4 -models, and Naish’s arrow is generalised as \sqsupseteq^4 (treating both as connectives), with the following truth table:

\sqsupseteq^4	u	t	f	i
u	t	f	f	f
t	t	t	f	f
f	t	f	t	f
i	t	t	t	t

Proposition 2

I is a \sqsupseteq^4 -model of P iff $\Phi_P(I) \sqsubseteq I$.

Proof

I is a \sqsupseteq^4 -model iff, for every head grounding (H, B) of P , $I(B) \sqsubseteq I(H)$. This is equivalent to stating that if I makes B true then I makes H true, and also, if I makes B false then I makes H false. But this is the case iff $\Phi_P(I) \sqsubseteq I$, by the definition of Φ_P . \square

It is easy to see that if I is a \sqsupseteq^3 -model of P then I is a \sqsupseteq^4 -model of P . However, the converse is not necessarily true, so the results of Naish (2006) cannot be used to show properties of four-valued models. However, such properties can be proved directly, using properties of the lattice of interpretations.

Proposition 3

The least \sqsubseteq^4 -model of P is $\text{lfp}(\Phi_P)$.

Proof

By the definition of \sqsubseteq^4 -model, I is a \sqsubseteq^4 -model iff $\Phi_P(I) \sqsubseteq I$. Since Φ_P is monotone, the Knaster-Tarski theorem (Tarski 1955) establishes $\text{lfp}(\Phi_P)$ as the least I such that $\Phi_P(I) \sqsubseteq I$. Hence $\text{lfp}(\Phi_P)$ is the least \sqsubseteq^4 -model of P . \square

For reasoning about partial correctness, the relationship between truth values in an interpretation and operational behaviour is crucial.

Theorem 1

If $I \sqsubseteq^4 \text{lfp}(\Phi_P)$ then no **t** atoms in I can finitely fail, no **f** atoms in I can succeed, and no **u** atoms in I can finitely fail or succeed.

Proof

The least fixed point in the four-valued case is the same as the least fixed point in the three-valued case. Hence (Kunen 1987) finitely failed atoms are **f** in $\text{lfp}(\Phi_P)$, successful atoms are **t** in $\text{lfp}(\Phi_P)$, and **u** atoms in $\text{lfp}(\Phi_P)$ must loop. From the \sqsubseteq ordering, an atom mapped to **f** by I can only be mapped to **f** or **u** by $\text{lfp}(\Phi_P)$. Similarly, atoms which I maps to **t** can only be mapped to **t** or **u** by $\text{lfp}(\Phi_P)$, and **u** atoms can only be mapped to **u**. \square

Corollary 1

If I is a \sqsubseteq^4 -model of P then no **t** atoms in I can finitely fail, no **f** atoms in I can succeed and no **u** atoms in I can finitely fail or succeed.

Proof

From Theorem 1 and Proposition 3. \square

These results about the behaviour of **t** and **f** atoms are essentially the two soundness theorems, for finite failure and success, respectively, of Naish (2006). The result for **u** atoms is new. The relationship between the (idealised) operational semantics and various forms of four-valued model-theoretic semantics can be summarised by the following Table (the last row summarises Corollary 1). It is a refinement of Table 1 of Naish (2006), which is the same except it uses three-valued models and conflates **i** and **u**.

operational	succeed	loop	fail
least $=^4$ -model	t	u	f
any $=^4$ -model	t	t/u/i/f	f
any \sqsubseteq^4 -model	t/i	t/u/i/f	i/f

Consider again the four programs depicted in Figure 3. Table 1 describes seven interpretations for these programs. I_0 is the typically intended interpretation, with inadmissibility of `eq_diff/4` defined as before, `sub(A,B,C)` inadmissible if **A** or **B**

	I_0	I_1	I_2	I_3	I'_3	I''_3	I_4
<code>eq_diff(s(0),0,s(s(0)),s(0))</code>	t	t	t	t	t	t	t
<code>eq_diff(s(0),0,0,0)</code>	f	f	f	f	f	f	u
<code>eq_diff([],[],[],[])</code>	i	f	f	t	t	t	t
<code>eq_diff([],0,[],0)</code>	i	f	t	u	f	t	u
<code>eq_diff(s(0),0,0,s(0))</code>	i	f	f	f	f	f	u
<code>eq_diff(0,s(0),0,s(0))</code>	i	f	f	u	f	t	u
P1 least model		✓					
P2 least model			✓				
P3 least model				✓			
P4 least model							✓
P1 $=^4$ -model		✓					
P2 $=^4$ -model			✓				
P3 $=^4$ -model				✓	✓	✓	
P4 $=^4$ -model							✓
P1 \sqsupseteq^4 -model	✓	✓					
P2 \sqsupseteq^4 -model	✓		✓				
P3 \sqsupseteq^4 -model	✓			✓	✓	✓	
P4 \sqsupseteq^4 -model	✓			✓	✓	✓	✓

Table 1. Seven interpretations of programs P1–P4 from Figure 3

are not natural numbers or $\mathbf{B} > \mathbf{A}$, and other atoms partitioned into **t** and **f** in the intuitive way. I_1 , I_2 , I_3 and I_4 are the least four-valued models of the programs $P1$ – $P4$, respectively. The truth values in these interpretations also align with the operational behaviour of the atoms in Prolog. I'_3 and I''_3 are the same as I_3 except that atoms which are **u** in I_3 are **f** and **t** in I'_3 and I''_3 , respectively. The top section of Table 1 gives the truth values of several representative atoms for each interpretation; we assume the existence of constant `[]` in the set of function symbols to show behaviour of “ill-typed” atoms. I_1 , I'_3 and I''_3 are two-valued, with I_1 the least two-valued model (using the truth ordering) of $P1$ and I'_3 the least two-valued model of both $P3$ and $P4$.

The later parts of Table 1 give which of these interpretations are certain kinds of four-valued models for the different programs. The four least models of the respective programs are distinct, reflecting the different behaviours. I'_3 and I''_3 are not the least model of $P3$ but they are $=^4$ -models.

Note carefully that the intended interpretation, I_0 , is a \sqsupseteq^4 -model of all programs. For the four interpretations shown which are \sqsupseteq^4 -models of $P3$, we have $I_0 \sqsupseteq I'_3 \sqsupseteq I_3$ and $I_0 \sqsupseteq I''_3 \sqsupseteq I_3$ with I'_3 and I''_3 incomparable in the information order. $P4$ also has all these interpretations as models, along with I_4 , which is below I_3 .

Also note how we use the two non-classical values in $\mathbf{4}$, that is, **u** and **i**, for quite distinct purposes. The two- and three-valued approaches to semantics do not allow such a complete picture of how these programs behave, along with the ways they can be viewed by programmers.

The use of \sqsupseteq^4 -models allows simple and intuitive verification of partial correctness of all programs but does not distinguish between total correctness ($P1$ – $P3$) and only partial correctness ($P4$). However, even analysis of least models does not guarantee total correctness for Prolog programs because alignment of truth values

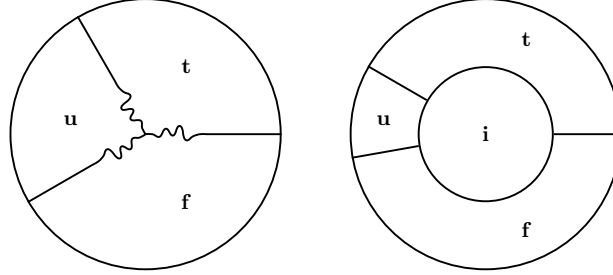
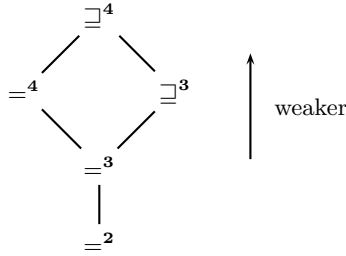
Fig. 4. Least vs typical intended \sqsubseteq^4 -model

Fig. 5. Relationship between model definitions

and behaviour assumes fairness of the search strategy (for success) and the computation rule (for finite failure) and non-floundering whenever negation is present. For example, if we reverse the order of the two sub-goals in the `sub/3` definition in *P3* then with Prolog's normal left to right computation rule, *P3* behaves the same as *P4* for the atoms shown.

Figure 4 gives a graphical representation of how the least model of a program compares with a typical intended model. In the least model, no atoms are *i*, and (ideally) there is a correspondence between the truth values of atoms and their behaviour. However, the distinction between these categories can be subtle and un-intuitive (hence the wiggly lines). For example, the atom `eq_diff([],0,[],0)` may succeed, finitely fail or loop, depending on how `sub/3` is coded. In a typical intended interpretation there are atoms which are *i* (they may have any other truth value in the least model). This allows the distinction between the categories to be more intuitive and allows a single interpretation to be a model of many different programs with different behaviours for the *i* atoms. The set of *u* atoms in a typical intended interpretation is a subset of the *u* atoms in the minimal model (often it is the empty set, which corresponds to a three-valued model of Naish (2006)). Atoms which are *u* in the minimal model can have any truth value in the intended model.

Figure 5 shows the relationship between the five different definitions of a model we have considered. Any interpretation which is a model according to one definition is also a model according to all definitions above. Weaker definitions of models allow more flexibility in how we think of our programs, yet still guarantee partial correctness.

7 A “model meet” property

With the classical logic approach for definite clause programs, we have a useful model intersection property: if M and N are (the set of true atoms in) models then $M \cap N$ is (the set of true atoms in) a model. Proposition 1 of Naish (2006) generalises this result using the truth ordering for three-valued interpretations, and Proposition 2 of Naish (2006) gives a similar result which mixes the truth and information orderings. However, none of these results hold for logic programs with negation. Here we give a new analogous result, using the information ordering, which holds even when negation is present. This will be utilised in our discussion of modes in Section 9.

Proposition 4

If M and N are \sqsupseteq^4 -models of program P then $M \sqcap N$ is a \sqsupseteq^4 -model of P .

Proof

Assume M and N are \sqsupseteq^4 -models of P . By Proposition 2, $\Phi_P(M) \sqsubseteq M$ and $\Phi_P(N) \sqsubseteq N$, since M and N are models. By monotonicity, $\Phi_P(M \sqcap N) \sqsubseteq \Phi_P(M) \sqsubseteq M$, and $\Phi_P(M \sqcap N) \sqsubseteq \Phi_P(N) \sqsubseteq N$. It follows that $\Phi_P(M \sqcap N) \sqsubseteq M \sqcap N$, so by Proposition 2, $M \sqcap N$ is a model of P . \square

For example, with the models of $P3$ in Table 1, $I3' \sqcap I3'' = I3$. The corresponding result does not hold for $=^4$ -models. Consider the following program:

```
p :- p.
q :- q.
r :- p ; q ; s.
s :- p ; q ; not r.
```

Let M be the interpretation which maps (p,q,r,s) to (t,f,t,t) , respectively, and let N be the interpretation (f,t,t,t) . Both M and N are $=^4$ -models. The meet, $M \sqcap N$, is (u,u,t,t) but Φ_P applied to this interpretation is (u,u,t,u) . So $M \sqcap N$ is a \sqsupseteq^4 -model but not a $=^4$ -model. (This example also shows that Φ_P , while monotone, is not in general an increasing function.)

8 Program analysis

This section and the three that follow it present several applications of the four-valued semantics we have introduced. We hope to convince the reader that there are numerous situations in which four-valued logic is the natural setting for reasoning about logic programs and their behaviour, and that \sqsupseteq^4 -models in particular can play an important role.

Four-valued logic provides a convenient setting for static analysis of logic programs. The reason is that program analysis almost always is concerned with run-time properties that are undecidable, so some sort of approximation is needed, to guarantee finiteness of analysis. As an example, we show how the program analysis framework proposed by Marriott and Søndergaard (1992) generates four-valued interpretations of the kind we have discussed.

Many program analyses for logic programs try to detect how logic variables are being used or instantiated. The well-known T_P function and Fitting's Φ_P function yield ground atomic formulas, and so semantic definitions based on these functions are not ideal as a basis for static analysis which intends to express what happens to variables at runtime. The s -semantics (Falaschi et al. 1988; Bossi et al. 1994) is a non-ground version of the T_P semantics. The s -semantics of a program P is a set S_P of possibly non-ground atoms, with the property that (a) the ground instances of the atoms in S_P give precisely the least Herbrand model of P , and (b) the computed answer substitutions (Lloyd 1984) for a query Q can be obtained by solving Q using the (potentially infinite) set S_P . Letting \mathcal{A} denote the set of atomic formulas, the s -semantics of P is defined as the least fixed point of an “immediate consequences” operator $T_P^v : \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathcal{A})$. More precisely,

$$T_P^v(I) = \left\{ h\theta \mid \begin{array}{l} C \equiv h : - b_1, \dots, b_n \in P, \langle a_1, \dots, a_n \rangle \ll_C I, \\ \theta = mgu(\langle b_1, \dots, b_n \rangle, \langle a_1, \dots, a_n \rangle) \end{array} \right\}$$

where $\langle a_1, \dots, a_n \rangle \ll_C I$ expresses that a_1, \dots, a_n are variants of elements of I renamed apart from C and from each other, and mgu gives the most general unifier of two (sequences of) expressions. As an example, for the append program,

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

the least fixed point of T_P^v is

$$\{\text{append}([x_1, \dots, x_n], ys, [x_1, \dots, x_n | ys]) \mid n \geq 0\}$$

Codish and Søndergaard (2002) give an account of the role of various (goal-directed as well as goal-independent) semantics, including the s -semantics, for the analysis of logic programs.

Marriott and Søndergaard (1992) use Fitting's three-valued semantics as a basis for defining static analyses which over-estimate both a given program's success set and its finite failure set. Fitting's Φ_P operator generates pairs (S, F) of sets of ground atoms, with the reading that every atom in S succeeds and every atom in F finitely fails, and it only allows for pairs that satisfy $S \cup F = \emptyset$. That is, there are three cases for an atom: It can be contained in S (have the value **t**), it can be contained in F (have the value **f**), or it can be absent from both (have the value **u**). In the “approximate” version of Marriott and Søndergaard (1992), S and F are allowed to share atoms. In the parlance of the present paper, these atoms are given the value **i**, though in the context of analysis it means “don't know” rather than “don't care”.

There are different ways in which we can guarantee finiteness of the static analysis. One of the approaches used by Marriott and Søndergaard (1992) is to never generate terms beyond a fixed, finite depth. We define the *depth* of a variable or a constant to be 1, and for other terms we define depth inductively:

$$\text{depth}(t) = 1 + \max \{ \text{depth}(u) \mid u \text{ is a proper subterm of } t \}.$$

The idea in *depth- k analysis* is that a term with a depth greater than k can be

that the program fails unexpectedly, leading to rather tedious analysis of the complex execution in order to uncover the mistake. One approach to avoid some runtime error diagnosis is to impose additional discipline on the programmer, generally restricting programming style somewhat, to allow the system to statically classify certain programs as incorrect. Various systems of “types” and “modes” have been proposed for this. An added benefit of some such systems is that they help make implementations more efficient. Here we discuss systems of this kind at a very high level and argue that four-valued interpretations potentially have a role in this area, particularly in mode systems such as that of Mercury (Somogyi et al. 1995).

Type systems typically assign a type (say, Boolean, integer, list of integers) to each argument of each predicate. This allows each variable occurrence in a clause to also be assigned a type. One common error is that two occurrences of the same variable have different types. For example, consider a predicate `head` which is intended to return the head of a list of integers but is incorrectly defined as: `head([_ | Y], Y)`. The first occurrence of `Y` is associated with the type “list of integers” and the other is associated with type “integer”. If `head` is called with both arguments instantiated to the expected types, it must fail. But `head` can succeed if it is called in different ways. For example, with only the first argument instantiated it will succeed, albeit with the wrong type for the second argument (and this in turn may cause a wrong result or failure of a computation which calls `head`).

Type systems can be refined by considering the “mode” in which predicates are called, or dependencies between the types of different arguments. This can allow additional classes of errors to be detected. For example, we can say the first argument of `head` is expected to be “input” and the second argument can be “output”. Alternatively (but with similar effect), we could say if the first argument is a list of integers, the second should be an integer. To see that mode information transcends type information, consider the (incorrect) definition `head([_ | Y], X)`. Here there is a consistent assignment of types to variables, but it does not satisfy the stipulated mode/type-dependency constraint. One high level property of several mode systems is that if input arguments are well typed then output arguments will be well typed for any successful call. In fact, a stronger property is desirable: the whole successful derivation should be well typed (otherwise we have a very dubious proof). Typically, well typed inputs in a clause head imply well typed inputs in the body, which implies well typed outputs in body, which implies well typed outputs in the head. This idea is present in the “directional types” concept (Aiken and Lakshman 1994; Boye and Małuszynski 1995), the mode system of Mercury (Somogyi et al. 1995), and the view of modes proposed in Naish (1996). Here we show the relevance of four-valued interpretations to this idea, ignoring the details of what constitutes a type (which differs in the different proposals) and what additional constraints are imposed (neither Mercury nor directional types support cyclic dataflow, and Mercury has additional interactions between types, modes and determinism).

We will present a mode system inspired by that of Mercury. Mercury allows types to be defined using `type` declarations and declared for predicate arguments using `pred` declarations; we adopt this verbatim. Mercury modes are declared using `mode` declarations, which also declare determinism (the range of possible numbers of so-

```

:- pred rev(list(T), list(T)).
:- mode rev(in, out) and (out, in).
rev([], []).
rev([H|T], R) :- rev(T, L), append(L, [H], R).

```

Fig. 6. Naive reverse with a group of two modes

lutions). We propose similar mode declarations which allow additional refinements. Determinism information could also be added but we ignore this aspect here. Similarly, we ignore issues surrounding negation. Mercury also supports an **error/1** primitive which results in abnormal termination if called. It allows more precise static analysis of modes (and of determinism) and we also adopt it.

Type and mode declarations document some aspects of how predicates are intended to be used and how they are intended to behave. We define interpretations which are consistent with these documented intentions. We assume there is a notion of well typedness for each argument of each predicate in program P .

Definition 6 (Mode, mode group, mode interpretation and moded program)

A *mode* for predicate P is an assignment of “input” or “output” to each of P ’s argument positions. A *mode interpretation* of P with a given mode m , $MI(P, m)$, is a four-valued interpretation M such that the value of (ground) atom A in M is

- **u**, if the predicate is **error/1**, and otherwise:
- **t**, if all arguments are well-typed,
- **i**, if some input argument (according to m) is ill-typed, and
- **f**, if all input arguments are well typed but some output argument is ill-typed.

A *mode group* is a set of modes for a predicate. A *mode interpretation* of P with a mode group $\{m_1 \dots m_k\}$ is $\prod_{1 \leq i \leq k} MI(P, m_i)$. A *moded program* is a program with a mode group defined for each predicate. Mode interpretations for moded programs are defined in the obvious way.

Note that the assignments **u** and **t** are independent of the mode(s).

For a mode group, an atom is **i** where there is no mode in the group for which all input are well typed. Changing the mode(s) of a predicate so it can be used in more flexible ways corresponds to changing the truth value of some atoms from **i** to **f**. This makes the mode interpretation more precise (lower in the information order). Asymmetry between **t** and **f** arises because mode analysis must “assume the worst” with respect what can succeed, local variables in clauses are existentially quantified in the body, and negated literals do not bind variables which appear in the rest of the clause body. Figure 6 illustrates the syntax we use for defining the modes of a predicate—mode groups are formed using the keyword “and”. The Mercury equivalent is to use two separate mode declarations.

Mercury uses the notion of an *implied mode*—a mode m implies all modes whose output arguments are a subset of those in m . For example, the mode **(in, in)** is implied by either of the two declared modes for **rev/2**. The next proposition says that mode interpretations are invariant under addition of implied modes.

Proposition 5

The mode interpretation of predicate P with modes $\{m_1 \dots m_k\}$ is the same as that for P with modes $\{m_1 \dots m_k, m'\}$ if the outputs of m' are a subset of the outputs of some m_j , $1 \leq j \leq k$.

Proof

For each j we have that $MI(P, m') \sqsubseteq MI(P, m_j)$, since each input argument of m_j is an input argument of m' . So $MI(P, m') \sqsubseteq \prod_{1 \leq j \leq k} MI(P, m_j)$ and thus $MI(P, m') \sqcap \prod_{1 \leq j \leq k} MI(P, m_j) = \prod_{1 \leq j \leq k} MI(P, m_j)$. \square

If a mode interpretation of a moded program P is a \sqsubseteq^4 -model this gives us the high level property discussed earlier:

Proposition 6

If a mode interpretation M of a moded program P is a \sqsubseteq^4 -model and A is a successful atom which, for some mode of the predicate, has all input arguments well typed, then A has all arguments well typed.

Proof

By Corollary 1, since M is a \sqsubseteq^4 -model and A succeeds, A must be **t** or **i** in M . By the definition of mode interpretations, since A is not **f** and all input arguments are well typed for some mode, all output arguments must be well typed as well. \square

For the stronger property to hold (the whole derivation being well-typed), the mode interpretation being a \sqsubseteq^4 -model is not sufficient. A definition can have a **t** head and a body which is a disjunction of a **t** atom which loops and an **i** atom which succeeds: although the interpretation is a \sqsubseteq^4 -model, the only successful derivation uses the inadmissible disjunct. To prevent such cases we impose an extra condition on each disjunct in the body of a definition (or each clause in a Prolog program) rather than the body of the definition as a whole:

Definition 7 (Well-moded)

A moded program P is *well-moded* with respect to mode interpretation M if M is a \sqsubseteq^4 -model of P and for each head grounding of a definition $(H, \exists W[C_1 \vee \dots \vee C_k])$ where $M(H) = \mathbf{t}$, $M(\exists W C_j) \neq \mathbf{i}$, $1 \leq j \leq k$.

In practice, it seems that the additional constraint rarely makes a difference. In the examples we discuss below, whenever the interpretation is a \sqsubseteq^4 -model, the program is well-moded with respect to the interpretation.

Lemma 1

If program P is well-moded, with mode interpretation M and atom A , with $M(A) = \mathbf{t}$, succeeds, then A is well typed and there is a ground predicate definition instance $(A, \exists W[C_1 \vee \dots \vee C_k])$ such that all literals in some C_j succeed and are assigned **t** and all positive literals in C_j are well typed.


```

% Extracts head of list. Has exactly one solution for all
% (admissible) calls. nonempty_head([],_) is inadmissible.
:- pred nonempty_head(list(T), T).
:- mode nonempty_head(in, out).
nonempty_head([H|_], H).

% Extracts head of list. Has exactly one solution for all
% (normally terminating) calls. checked_head([],_) throws an error.
:- pred checked_head(list(T), T).
:- mode checked_head(in, out).
checked_head([], _) :- error("head of empty list").
checked_head([H|_], H).

```

Fig. 7. Two versions of head for non-empty lists

Proof

Since M is a mode interpretation and $M(A) = \mathbf{t}$, all arguments are well typed. A successful disjunct C_j must exist; it cannot be \mathbf{i} since P is well-moded, so it must be \mathbf{t} (only \mathbf{i} and \mathbf{t} disjuncts can succeed, by Corollary 1). Similarly, no literal in C_j can be \mathbf{u} , so all literals in C_j must be \mathbf{t} , thus each positive literal in C_j must be well typed. \square

Theorem 2

If P is a well-moded program and A is a successful atom which is \mathbf{t} in the mode interpretation of P , then there is successful derivation of A in which all successful atoms are well typed.

Proof

By induction on the depth of the proof and Lemma 1. \square

Checking that a mode interpretation is a \sqsubseteq^4 -model (and the additional constraint holds) requires the kind of analysis used in other forms of mode checking. For example, consider again Figure 6. Assume the recursive clause for **rev**/2 uses mode **(in,out)** and assume **append**/3 has mode **(in,in,out)**. Mode analysis intuitively reasons that if **rev**/2 is called with the first argument well typed ($[H|T]$ is a list), the recursive call will be called with its input argument well typed (T is a list), thus in any successful call its output argument will be well typed (L is a list), the input arguments to **append**/3 will be well typed so its argument will be well typed (R is a list), so the head will be well typed. In other words, if we assume the head is \mathbf{t} or \mathbf{f} , we can find an instance of the body which is \mathbf{t} and the head must be \mathbf{t} . Thus there are no head clause instances of the form $\mathbf{t}:-\mathbf{i}$, $\mathbf{f}:-\mathbf{i}$, $\mathbf{f}:-\mathbf{t}$ or $\mathbf{t}:-\mathbf{f}$ (and the head cannot be \mathbf{u} since the predicate is not **error**/1), so the mode interpretation is a \sqsubseteq^4 -model.

Sometimes the most natural intended interpretation has certain atoms assigned \mathbf{i} , but static mode analysis is unable to conclude such atoms are never called. In Mercury this issue arises more with determinism, when analysis is unable to conclude that a given atom always succeeds. Using **error**/1 allows us to make static

analysis more flexible by making the code more verbose and introducing some run-time checking. As an example of this, consider extracting the head of a list. There are situations where we expect this operation to be called on non-empty lists only (so the computation has exactly one solution, or is “det” in Mercury terminology). Figure 7 gives two codings. The first cannot easily be checked statically. Our proposed mode system has no way to declare well-typed atoms as inadmissible, so clauses of the form $\mathbf{t}:-\mathbf{i}$ in our intended interpretation may be possible. Similarly, Mercury cannot determine the code is “det”. The second is acceptable for Mercury and also safe for our mode system. By changing the intended interpretation of `checked_head([],_)` from \mathbf{i} to \mathbf{u} , the mode interpretation (where it is assigned \mathbf{t}) becomes a safe approximation to the intended interpretation. Thus, having mode interpretations that can distinguish \mathbf{i} from \mathbf{u} can be helpful.

There is one more feature of the mode system we propose—allowing more than one mode group per predicate. This feature is not supported in any other mode systems. Separate mode groups are declared using the keyword “also”. In the following example each group has a single mode, but in general we can use a mixture of “and” and “also”, with the former binding more tightly.

```
:- mode rev(in, out) also (out, in).
```

For the external view of a predicate, for example, how modes approximate the behaviour of a non-recursive call to a predicate, “also” is treated identically to “and” (the meet of the mode interpretations is used). However, for the internal view of a predicate and how its definition is checked for well-modedness, we impose a stronger constraint. The definition must be well-moded with respect to *each* interpretation corresponding to a mode group. This implies it is also well moded with respect to the meet (we give the case of two mode groups; the generalisation to N mode groups is straightforward):

Proposition 7

If predicate P is well-moded with respect to mode interpretations MI_1 and MI_2 then it is well-moded with respect to M , where $M = MI_1 \sqcap MI_2$.

Proof

M is a \sqsubseteq^4 -model, by Proposition 4. Consider a head grounding of the definition of P , $(H, \exists W[C_1 \vee \dots \vee C_k])$. If $M(H) = \mathbf{t}$ then $MI_1(H) = \mathbf{t}$ or $MI_2(H) = \mathbf{t}$, so $MI_1(C_j) \neq \mathbf{i}$ or $MI_2(C_j) \neq \mathbf{i}$, so $M(C_j) \neq \mathbf{i}$, for $1 \leq j \leq k$. \square

For sets of mutually recursive predicates there must be some set of mode interpretations S , the predicates must be well-moded with respect to each element of S , and S must have a mode interpretation for each mode group in each of the predicates (each mode group of a predicate must be “covered” by at least one element of S).

Consider the naive reverse example with the mode declaration above and assume `append/3` has modes $(\mathbf{in}, \mathbf{in}, \mathbf{out})$ and $(\mathbf{out}, \mathbf{in}, \mathbf{in})$. There are two \sqsubseteq^4 -models corresponding to the mode interpretations for modes $(\mathbf{in}, \mathbf{out})$ and $(\mathbf{out}, \mathbf{in})$, respectively, and a third \sqsubseteq^4 -model which is the meet. However, if we swap the arguments in the recursive call to `rev/2`, the meet is still a \sqsubseteq^4 -model but the other two

```

:- pred rev_ra(list(T), list(T)).
:- mode rev_ra(in, out) also (out, in).
rev_ra([], []).
rev_ra([H|T], R) :- rev_rb(L, T), append(L, [H], R).

:- pred rev_rb(list(T), list(T)).
:- mode rev_rb(in, out) also (out, in).
rev_rb([], []).
rev_rb([H|T], R) :- rev_ra(L, T), append(L, [H], R).

```

Fig. 8. Mutually recursive reverse with complementary modes

mode interpretations are *not* \sqsubseteq^4 -models. Calling `rev/1` in mode `(in,out)` requires a recursive call in mode `(out,in)` and vice versa, so one mode alone is not sufficient and mode checking with the “also” mode declaration would fail. Mode declarations with “also” are stronger than those with “and”; they tell us more about the set of \sqsubseteq^4 -models. The additional expressiveness can be used to detect more errors (for example, if the arguments were swapped accidentally and the stronger mode declaration was used).

The version of reverse with the arguments swapped can be specialised to two mutually recursive predicates, shown in Figure 8. This has three \sqsubseteq^4 -models: one with mode `(in,out)` for `rev_ra/2` and mode `(out,in)` for `rev_rb/2`, another with mode `(out,in)` for `rev_ra/2` and mode `(in,out)` for `rev_rb/2`, and the third is the meet. The program is well-moded with respect to all three and each mode group of each predicate is covered by one of these models.

Figure 9 gives another example of the additional expressive power. The mode declared for `fold_and3/2` is redundant: it has `(in,out)` and the implied mode `(in,in)`. However, even though `(in,in)` is weaker in some sense, and its corresponding mode interpretation is strictly higher in the information ordering, it is not a \sqsubseteq^4 -model. Calling `fold_and3/2` in mode `(in,in)` requires a recursive call in mode `(in,out)`. However, for `fold_and3a/2`, which computes the same thing, the code is well-moded with respect to each of the mode interpretations, as expressed by the “also”. The mode `(in,in)` does not rely on mode `(in,out)` and considerably better efficiency can be achieved, because it can be statically determined (by the Mercury compiler, for example) that no choice points are needed.

Precise analysis of declared types, modes, determinism, and so on, is useful for uncovering program errors statically and increasing efficiency of implementations. Such analysis distinguishes computations which (might) succeed from those which (must) fail. Most proposals also support methods to restrict the ways in which predicates should be used, for example, the input arguments should be well typed. The more advanced proposals also support forms of abnormal termination, such as `error/1`. The four-valued domain we use for the semantics of logic programs seems particularly well suited to this kind of analysis. In particular, we have demonstrated how type and mode declarations can be used to define four-valued interpretations and how \sqsubseteq^4 -models are an important device for checking correctness of these declarations.

```

:- type b ---> t ; f.           % Boolean
:- type k3 ---> t3 ; f3 ; i3.  % Kleene

% 'and' where third truth value means maybe true, maybe false
:- pred and3(k3, b, b).
:- mode and3(in, in, out).
and3(i3, _, f).
and3(i3, t, t).
and3(f3, _, f).
and3(t3, B, B).

% 'and3' of each value in a list
:- pred fold_and3(list(k3), b).
:- mode fold_and3 (in, out) and (in, in). % latter is redundant
fold_and3([], t).
fold_and3([f3|_], f).
fold_and3([B3|B3s], R) :- fold_and3(B3s, R1), and3(B3, R1, R).

:- pred fold_and3a(list(k3), b).
:- mode fold_and3a (in, out) also (in, in).
fold_and3a([], t).
fold_and3a([f3|_], f).
fold_and3a([i3|_], f).
fold_and3a([i3|B3s], t) :- fold_and3a(B3s, t).
fold_and3a([t3|B3s], R) :- fold_and3a(B3s, R).

```

Fig. 9. Illustration of “and” versus “also” in modes

$$\forall s \forall s' \text{subset}(s, s') \leftrightarrow \forall e (\text{member}(e, s) \rightarrow \text{member}(e, s'))$$

```

subset([], _).
subset([E|SS], S) :- member(E, S), subset(SS, S).

member(E, [E|_]).
member(E, [_|S]) :- member(E, S).

list([]).
list([_|S]) :- list(S).

```

Fig. 10. First-order logic specification and Prolog definition of `subset/2`

10 Formal Specifications

In the early days of logic programming there was considerable interest in the relationship between specifications (particularly formal specifications written in classical first order logic) and logic programs (Clark and Sickel 1977; Hogger 1981; Sato and Tamaki 1984; Kowalski 1985). This work generally overlooked what we here have called inadmissibility. For example, Figure 10 shows a specification and Prolog implementation of the `subset/2` predicate given by Kowalski (1985), where sets are represented as lists and *member* is the Prolog list membership predicate. Kowalski (1985) shows that the implementation is a logical consequence of the

specification. That is to say, the program P which defines `subset` is sound with respect to the specification S : for all queries Q , if $P \models Q$ then $S \models Q$. However, `subset(true,42)` is true according to the specification, which is counter-intuitive, to say the least. If the specification is modified to restrict both arguments to be lists, the program is no longer a logical consequence (the program has `subset([],42)` as a consequence but `subset([],42)` is no longer a consequence of the specification). When negation is also considered, or even the fact that logic programs implicitly define falsehood of some atoms, early approaches relating formal specifications and logic programs based on classical logic seem unworkable.

With our approach it is natural to identify specifications with four-valued interpretations. Our “intended interpretations” are essentially specifications, albeit informal ones which exist only in the mind of programmers. However, we can also design formal specification languages where the meaning of a specification is a single four-valued interpretation. We propose such a language now. Although we can never be sure that a formal specification accurately captures our intentions (as Kowalski’s specification above shows), and fully automated verification is bound to be intractable in general, cross-checking between a specification and code can give us additional confidence in the correctness of our code. In the design of our specification language we aim to utilise classical logic as far as possible, while allowing the flexibility of all four values. Underspecification is supported by declaring preconditions as well as postconditions.

Definition 8 (Specification)

A specification is a well formed formula (wff) Δ , a set of distinct atoms A_i in most general form, a precondition wff α_i for each A_i and a postcondition wff ω_i for each A_i .

For example, a precondition and postcondition of `subset/2` could be defined using syntax exemplified below (which could be supported by just declaring the three keywords as operators in NU-Prolog or Mercury). In addition, Δ would define the predicates `member/2` and `list/1` using a syntax close to traditional first order logic, or Prolog syntax could be used as shorthand for the Clark completion, for example.

```
predicate subset(SS, S)
precondition list(S), list(SS)
postcondition all [E] (member(E, SS) => member(E, S)).
```

Definition 9 (Meaning of a specification)

The meaning of a specification is a four-valued interpretation for the A_i predicates, such that each ground atom $A_i\theta$ is

- **i**, if precondition $\alpha_i\theta$ is false in any classical model of Δ , otherwise
- **t**, if postcondition $\omega_i\theta$ is true in all classical models of Δ ,
- **f**, if postcondition $\omega_i\theta$ is false in all classical models of Δ , and
- **u** otherwise.

With this, a `subset/2` atom which has some non-list argument is given the value **i**, and no `subset/2` atom gets the value **u**. The other `subset` atoms are partitioned into **t** and **f** in the intuitive way. Thus the counter-intuitive consequences of Kowalski’s specification are mapped to **i** rather than **t**.

Kowalski and others attempted to relate the meanings of formal specifications and programs via the truth ordering. In our approach we relate them via the information ordering. A program is correct with respect to a specification if and only if the meaning of the specification is greater than or equal to (\sqsupseteq^4) the least \sqsupseteq^4 -model of the program. The meaning of the specification being a \sqsupseteq^4 -model of the program is a sufficient condition for this and Theorem 1 gives the partial correctness results. For example, the meaning of the `subset/2` specification is a \sqsupseteq^4 -model of the program. There can be different logic programs, with different behaviours, which are correct according to a specification—they can be seen as refinements of the specification. If the specification is not a \sqsupseteq^4 -model of the program, the program may succeed or finitely fail in ways which are inconsistent with the specification (wrong answers or missing answers).

As we develop an implementation from an initial high level specification, we generally move lower in the information order. For example, we may find it useful to strengthen the specification above so `subset/2` can be used in more flexible ways in our system. Moving the `list(SS)` constraint from the precondition to the postcondition is like changing the mode declaration for `subset/2` from `(in,in)` to `(out,in)`. The meaning of the stronger specification is lower in the order, with some previously **i** atoms such as `subset(abc, [])` now being **f**. As we proceed from specification to code we go lower still: `subset([], 42)` is **i** in both specifications but **t** in the least \sqsupseteq^4 -model of the program. We believe our approach of having a complete lattice in the information order can provide a simple, elegant and accurate view of the relationship between specifications and programs.

Our proposed specification language is inspired in part by the VDM-SL specification language, which has preconditions but is based on the functional programming paradigm (functions are specified rather than predicates). The underlying theory is the logic of partial functions, LPF (Barringer et al. 1984; Jones and Middelburg 1994), with three truth values. Although specifications have preconditions, the primary use of the non-classical truth value is to represent undefined, or **u**—the meaning of a recursively defined function is given by the least fixed point of the definition. There is no separate truth value to represent unspecified, or **i**. Once again we contend that these semantically distinct notions are best represented by distinct truth values.

11 Declarative debugging

The semantics of Naish (2006) is closely aligned with declarative debugging (introduced in Shapiro (1983)) and the term “inadmissible” comes from this area (Pereira 1986). The Naish semantics gives a formal basis for the three-valued approach to declarative debugging of Naish (2000) (using **t**, **f**, and **i**) as applied to Prolog. Given a goal whose behaviour is inconsistent with the intended three-valued interpretation (it has a wrong or missing answer), the debugger identifies some part

of the code (such as a clause instance) which demonstrates that the intended interpretation is not a \sqsubseteq^3 -model. As we have demonstrated in Section 6, four values allow us to express programmer intentions more precisely than three. In this section we sketch how four-valued interpretations can be supported by declarative debuggers.

The declarative debugging scheme represents the computation as a tree; sub-trees represent sub-computations. Each node is classified by an oracle as correct, erroneous or inadmissible. The debugger searches the tree for a *buggy* node, which is an erroneous node with no erroneous children. If all children are correct it is called an *e-bug*, otherwise (it has an inadmissible child) it is called an *i-bug*. Every finite tree with an erroneous root contains at least one buggy node and finding such a node is the job of a declarative debugger. To diagnose wrong answers in Prolog a proof tree (Lloyd 1984) is used to represent the computation. Nodes containing **t**, **f** and **i** atoms are correct, erroneous and inadmissible, respectively. To diagnose computations that *miss* answers, a different form of tree is used, and nodes containing finitely failed **t**, **f** and **i** atoms are erroneous, correct, and inadmissible, respectively. Naish (2000) also spells out how to deal with some additional complexities which arise, such as non-ground wrong answers and computations which return some but not all correct answers; we skip the details here. Buggy nodes correspond to instances of definitions of the form **t**:-**f**, **f**:-**t**, **t**:-**i** or **f**:-**i**. The first two are e-bugs (the kind diagnosed by more conventional two-valued declarative debuggers); the last two are i-bugs.

Four-valued interpretations can be used in place of three-valued interpretations in this scheme, as follows. The debugging algorithm remains unchanged; only the way the oracle classifies nodes is modified. For wrong answer diagnosis, **u** is treated the same as **f**—a sub-computation which succeeds contrary to our intentions is erroneous. For missing answer diagnosis **u** is treated the same as **t**—a sub-computation which finitely fails contrary to our intentions is also erroneous. This simple generalisation of the three-valued scheme allows us to use four-valued interpretations and find bugs corresponding to instances of definitions where the head is **u** but the body is not.

For example, an atom such as `interpret("main:-main.")` may be considered admissible, since its argument is well-formed. However, it is not intended to terminate and if it succeeds (or finitely fails) we would like a tool to help debug it. With four values, we can say this atom is **u** and if the atom appears in the node of a proof tree, the node would therefore be considered erroneous and amenable to declarative debugging. The intended interpretation is not a \sqsubseteq^4 -model and the debugger is able to diagnose why.

Intuition may suggest the debugger would need four classes of nodes for the four truth values. However, the classes of nodes do not all correspond to truth values in the intended interpretation. They correspond to the *comparison* between the truth value in the intended interpretation and the observed behaviour (or the truth value in the least model of the program). Note that the observed behavior is two-valued in these uses of declarative debugging—the computation must succeed or finitely fail. Inadmissible nodes correspond to a comparison using \sqsubset (which only holds when the

intended value is **i**). Correct nodes correspond to $=$. Erroneous nodes correspond to incomparability for three-valued interpretations but they may also correspond to \sqsubset in the four-valued case. Thus four-valued interpretations add some flexibility to declarative debuggers with very little additional cost.

12 Computation and the information ordering

The logic programming paradigm introduced the view of computation as deduction (Kowalski 1980). Classical logic was used and hence computation was identified with the truth ordering. With Prolog programs viewed as Horn clauses, $:-$ is classical \leftarrow , or \geq in the truth ordering. We view the Prolog arrow as \sqsupseteq^4 , which naturally leads to identifying computation with the information ordering rather than the truth ordering. In this section we sketch this alternative view of the logic programming paradigm. The information ordering holds if we compare successive states of a computation using a correct program (that is, the intended interpretation is a \sqsupseteq^4 -model). Because $H \sqsupseteq B$ for each head grounding, replacing a subgoal by the body of its definition (a basic step in a logic programming computation) gives us a new goal which is lower (or equal) in the information ordering (see Proposition 8).

This view is obscured if we view Prolog computation as SLD derivations because SLD derivations include the “success continuation” of the current sub-goal but not the “failure continuation”—the alternatives which would be explored on backtracking. We view a computation state as a disjunction of (conjunctive) goals. This is equivalent to a *frontier* of nodes in an SLD tree rather than a single node (or a single goal in an SLD derivation). Free variables are those appearing in the top-level goal; other variables are existentially quantified. A computation step selects a node from the frontier (a disjunct), then selects a subgoal within it (a conjunct). For simplicity, we do not deal with negation here. A more detailed model of logic programming computation in this style would also include propagation of failure from unsatisfiable equations.

Definition 10 (Computation state, successor state)

A computation state S is a formula of the form $\exists V(D_1 \vee \dots \vee D_m)$, with each D_i a conjunction of literals $C_{i,1} \wedge \dots \wedge C_{i,m^i}$. Let $(C_{i,j}, \exists W(B_1 \vee \dots \vee B_n))$ be a head instance of a definition, with variables in W renamed so they are distinct from those in S . Let D' be $(C_{i,1} \wedge \dots \wedge C_{i,j-1} \wedge B_1 \wedge C_{i,j+1} \wedge \dots \wedge C_{i,m^i}) \vee \dots \vee (C_{i,1} \wedge \dots \wedge C_{i,j-1} \wedge B_n \wedge C_{i,j+1} \wedge \dots \wedge C_{i,m^i})$. Then $S' = \exists V \exists W(D_1 \vee \dots \vee D_{i-1} \vee D' \vee D_{i+1} \vee \dots \vee D_m)$ is a successor state of S .

Given a top-level Prolog goal, the intended interpretation gives a truth assignment for each ground instance. Subsequent resolvents can also be given a truth assignment for each ground instance of the variables in the top level goal (with local variables considered existentially quantified). As the computation progresses, the truth value assignment for each instance often remains the same, but can become lower in the information ordering.

Proposition 8

If S' is the successor state of S , θ is a grounding substitution for just the free variables in S (and S') and interpretation M is a \sqsubseteq^4 -model of the program, then $M(S\theta) \sqsubseteq^4 M(S'\theta)$.

Proof

Since variables in W are not in S , $S'\theta = (\exists V(D_1 \vee \dots D_{i-1} \vee (\exists W D') \vee D_{i+1} \dots \vee D_m)\theta$. Since variables in W are not in $C_{i,k}$ for $k \neq j$ and De Morgan's laws hold for $\mathbf{4}$, $D'\theta = (C_{i,1} \wedge \dots C_{i,j-1} \wedge \exists W(B_1 \vee \dots \vee B_n) \wedge C_{i,j+1} \dots \wedge C_{i,m})\theta$. Since M is a \sqsubseteq^4 -model, $M(C_{i,j}\theta) \sqsubseteq M(\exists W(B_1 \vee \dots \vee B_n)\theta)$. The result follows from the monotonicity of \wedge and \vee . \square

For example, consider the goal `implies(X,f)` (where `implies` was defined in Section 2). Our intended interpretation maps `implies(f,f)` to `t` and `implies(t,f)` to `f`, but may map `implies(42,f)` to `i`, if the first argument is expected to be input. After one step of the computation we have the conjunction `neg(X,U), or(U,f,t)` (where U is local to the computation and hence existentially quantified). If our intended interpretation allows any mode for `neg`, the instance where $X = 42$ is then mapped to `f`.

We believe that having a complete lattice using the information ordering provides an important and fundamental insight into the nature of computation. At the top of the lattice we have an element which corresponds to underspecification in the mind of a person. At the bottom of the lattice we have an element which corresponds to a the inability of a machine or formal system to compute or define a value. The transitions between the meanings we attach to specifications and correct programs, and successive execution states of a correct program, follow the information ordering, rather than the truth ordering.

13 Related work

Denecker et al. (2001) discuss the use of inductive definition in mathematical logic. They develop a general theory of induction over non-monotone operators, and at the same time provide strong justification for the well-founded semantics (Van Gelder et al. 1991; Fitting 1993) for logic programs with negation. The view of Denecker et al. (2001) is that recursive logic programs represent inductive definitions — the view to which we, with this paper, also subscribe. Denecker et al. (2001) is not concerned with intended semantics and specification, but the authors still make essential use of four-valued (as opposed to three-valued) logic, albeit primarily for reasons of technical convenience.

Arieli (Arieli 2002) similarly gives a fixed point characterisation of the meaning of logic programs. One aim is to provide a language that supports knowledge revision and reasoning with uncertainty. Arieli's logic programming language has two kinds of negation, namely explicit negation (\neg) and negation-by-failure (`not`). The proposed semantics allows for paraconsistency, that is, the handling of locally inconsistent information in a way that does not lead to the entire program being considered inconsistent. That context naturally leads to the use of Belnap's logic.

Loyer et al. (2004) are similarly concerned with an extended language. In this case, the language is that of “Fitting programs”, the kind of logic programs used by Fitting (1991a), with the usual connectives “duplicated” for the bilattice **4**. Loyer et al. (2004) extend Fitting’s work on reasoning in a distributed (or multi-agent) setting. The semantic framework they propose separates “hypotheses” from “facts” and is broad enough that, when restricted to Datalog programs, it generalises both Fitting’s “Kripke-Kleene” semantics (Fitting 1985) and the well-founded semantics (Van Gelder et al. 1991). The framework, which again is based on four-valued logic, provides what can be seen as a well-founded semantics for Fitting programs.

Many-valued logics have also long been advocated outside the logic programming community, but the take-up there has arguably been more limited. In Section 10 we briefly mentioned the aims and ideas of the Vienna Development Method (VDM). This school has long argued that since programs, functions, and procedures that are written in a Turing complete language may be partial, some sort of “logic for partial functions” is needed, and that such a logic necessarily is three-valued. As an extension, Arieli and Avron (1996; 1998) have argued the case for four-valued logic.

Starting with McCarthy (1963), many have argued in favour of many-valued logics in which connectives such as \wedge and \vee are no longer commutative. For example, in McCarthy’s logic, $\mathbf{t} \vee \mathbf{u}$ is equivalent to \mathbf{t} , but $\mathbf{u} \vee \mathbf{t}$ is equivalent to \mathbf{u} (whereas in K_3 it is \mathbf{t} as well). The lack of commutativity makes these connectives implementable in a *sequential* programming language, and it corresponds closely to how the connectives are defined in most modern programming languages. As an example of the use of many-valued logic with non-commutative conjunction, Barbuti et al. (1998) give a pure-Prolog semantics which is designed to closely mirror Prolog’s depth-first left-to-right evaluation strategy. The logic has four truth values, and the roles of \mathbf{u} , \mathbf{f} , and \mathbf{t} are conventional. However, the fourth value, denoted \mathbf{t}_u , is very different to \mathbf{i} . Its operational-semantics reading is that it stands for divergence preceded by success. Another example of the use of non-commutative conjunction is the three-valued logic proposed by Avron and Konikowska (2009) which combines K_3 (for reasoning about parallel constructs) with McCarthy’s logic (for reasoning about sequential constructs).

Morris and Bunkenburg (1998) are concerned with *program refinement* in the presence of partiality and non-determinism (in program statements and/or in specifications). They present a four-valued calculus over a language which includes a (non-monotone) “defined” predicate, a device also used in LPF.

Chechik et al. (2003) use Belnap’s 4-valued logic for the analysis of so-called mixed transition systems (Dams et al. 1997). Transitions in mixed transition systems carry a modality (may or must) with no assumption that a “must” transition also necessarily is a “may” transition. As a consequence, it is possible for a property to both hold and not hold.

Nishimura (2010) uses the simple 4-valued bilattice **4** in a variant of refinement calculus. Predicate transformers are developed for a small language of program statements, including an exception catching primitive, for use with abnormal pro-

gram behaviour (division by zero, say) as well as with explicit programmer-raised exceptions. Nishimura’s use of **4**, however, does not reflect a need to capture varying degrees of information content. Rather, four-valued logic is used to provide an elegant encoding trick. In Nishimura’s setting, $wp(S, \varphi_n, \varphi_e)$ expresses the weakest condition which, when it holds before program statement S , will ensure that, either, S terminates normally, making φ_n true, or else S terminates abnormally, making φ_e true. The four-valued lattice provides a convenient way of representing the four possible states of the pair $\langle \varphi_n, \varphi_e \rangle$.

The use of many-valued logic for reasoning about programs has also had its detractors who argue that abandoning classical logic complicates things, for insufficient gain. Gries and Schneider (1995) are concerned that three-valued logics abandon the law of the excluded middle, so that the schema $\varphi \vee \neg\varphi$ no longer is valid. They point out that, if $\mathbf{u} \Leftrightarrow \mathbf{u}$ is valid (and they insist that every instance of $\varphi \Leftrightarrow \varphi$ ought to be valid) then the bi-implication connective \Leftrightarrow fails to be associative, since we otherwise would have

$$\mathbf{f} \equiv \mathbf{t} \Leftrightarrow \mathbf{f} \equiv (\mathbf{u} \Leftrightarrow \mathbf{u}) \Leftrightarrow \mathbf{f} \equiv \mathbf{u} \Leftrightarrow (\mathbf{u} \Leftrightarrow \mathbf{f}) \equiv \mathbf{u} \Leftrightarrow \mathbf{u} \equiv \mathbf{t}$$

that is, we would have inconsistency. They conclude that three-valued logic is too complicated to use and favour staying instead with 2-valued logic by somehow side-stepping non-denoting terms. Problematic terms should be carefully prefixed to avoid non-denotation. For example, “ $y/y = 1$ ” should systematically be replaced by “ $y \neq 0 \Rightarrow y/y = 1$ ”.

To us it seems that Gries and Schneider (1995) ask for too much. It is only to be expected that the law of the excluded middle will be lost once we allow non-denoting terms in statements. And in the context of non-denoting terms, taking $\mathbf{u} \Leftrightarrow \mathbf{u}$ as valid would seem counter-intuitive. It is a stretch to consider the statement $n/0 = 42 \Leftrightarrow n/0 = 5$ valid, given that $42 \neq 5$. A far more natural approach is to consider that statement ill-defined, that is, being neither true nor false. As for the guarding of a formula φ by conditions that ensure all terms in φ are denoting, that is hardly a practical solution when the terms involved stem from a Turing complete language—in place of “ $y/y = 1$ ” consider being confronted with “ $f(y) = 1$ ”, where f has been given a (possibly complex) recursive definition.

14 Conclusion

Four-valued logic has previously been suggested as a tool for reasoning about program behaviour in the context of partiality, non-determinism and underspecification. In a logic programming context, it has been used for parallel and distributed programming, both as a language feature (Fitting 1991a) and an analysis tool (Palmer 1997). In this paper we have argued that four-valued logic provides a handle on many different situations that call for reasoning about logic programs, even when we restrict attention to sequential programming. The applications include program analysis, type and mode systems, formal specification, and declarative debugging. Moreover, a semantics based on four truth values turns out to be no more complex than one based on three.

Logicians have been aware of the limitations of formal systems since well before the invention of electronic computers. Gödel showed the impossibility of a complete proof procedure for elementary number theory, hence important gaps between truth and provability, and in any Turing-complete programming language there are programs which fail to terminate—undefinedness is unavoidable. Our awareness of the limitations of humans in their interaction with computing systems goes back even further. Babbage (1864) claims to have been asked by members of the Parliament of the United Kingdom, “Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out”? The term “garbage in, garbage out” was coined in the early days of electronic computing and concepts such as “preconditions” have always been important in formal verification of software—underspecification is also unavoidable in practice.

Using a special value to denote undefinedness is the accepted practice in programming language semantics. Using a special value to denote underspecification is less well established, but has been shown to provide elegant and natural reasoning about partial correctness, at least in the logic programming context. In this paper we have proposed a domain for reasoning about Prolog programs which has values to denote both undefinedness and underspecification—they are the bottom and top elements of a bilattice. This gives an elegant picture which encompasses both humans not making sense of some things and computers being unable to produce definitive results sometimes. The logical connectives Prolog uses in the body of clauses operate within the truth order in the bilattice. However, the overall view of computation does not operate in the truth order, it operates in the orthogonal “information” order.

References

- AIKEN, A. AND LAKSHMAN, T. K. 1994. Directional type checking of logic programs. In *Static Analysis*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer, 43–60.
- APT, K. R. AND BOL, R. N. 1994. Logic programming and negation: A survey. *Journal of Logic Programming* 19&20, 9–71.
- ARIELI, O. 2002. Paraconsistent declarative semantics for extended logic programs. *Annals of Mathematics and Artificial Intelligence* 36, 381–417.
- ARIELI, O. AND AVRON, A. 1996. Reasoning with logical bilattices. *Journal of Logic, Language and Information* 5, 25–63.
- ARIELI, O. AND AVRON, A. 1998. The logical role of the four-valued bilattice. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 118–126.
- AVRON, A. AND KONIKOWSKA, B. 2009. Proof systems for reasoning about computation errors. *Studia Logica* 91, 2, 273–293.
- BABBAGE, C. 1864. *Passages from the Life of a Philosopher*. Longman and Co., London.
- BARBUTI, R., DE FRANCESCO, N., MANCARELLA, P., AND SANTONE, A. 1998. Towards a logical semantics for pure Prolog. *Science of Computer Programming* 32, 145–176.
- BARRINGER, H., CHENG, J. H., AND JONES, C. B. 1984. A logic covering undefinedness in program proofs. *Acta Informatica* 21, 251–269.

- BELNAP, N. D. 1977. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, J. M. Dunn and G. Epstein, Eds. D. Reidel, 8–37.
- BLAIR, H. 1982. The recursion-theoretic complexity of the semantics of predicate logic as a programming language. *Information and Control* 54, 25–47.
- BOSSI, A., GABBRIELLI, M., LEVI, G., AND MARTELLI, M. 1994. The s-semantics approach: Theory and applications. *Journal of Logic Programming* 19&20, 149–197.
- BOYE, J. AND MALUSZYNSKI, J. 1995. Two aspects of directional types. In *Proceedings of the 12th International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, 747–761.
- CHECHIK, M., DEVEREUX, B., EASTERBROOK, S., AND GURFINKEL, A. 2003. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology* 12, 4, 371–408.
- CLARK, K. AND SICKEL, S. 1977. Predicate logic: A calculus for the formal derivation of programs. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*. 419–420.
- CLARK, K. L. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, 293–322.
- CODISH, M. AND SØNDERGAARD, H. 2002. Meta-circular abstract interpretation in Prolog. In *The Essence of Computation: Complexity, Analysis, Transformation*, T. Mogensen, D. Schmidt, and I. H. Sudborough, Eds. Lecture Notes in Computer Science, vol. 2566. Springer, 109–134.
- DAMS, D., GERTH, R., AND GRUMBERG, O. 1997. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems* 19, 2, 253–291.
- DENECKER, M., BRUYNOOGHE, M., AND MAREK, V. 2001. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic* 2, 4, 623–654.
- FALASCHI, M., LEVI, G., GABBRIELLI, M., AND PALAMIDESSI, C. 1988. A new declarative semantics for logic languages. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, 993–1005.
- FITTING, M. 1985. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming* 2, 4, 295–312.
- FITTING, M. 1988. Logic programming on a topological bilattice. *Fundamenta Informaticae* 11, 209–218.
- FITTING, M. 1989. Negation as refutation. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 63–70.
- FITTING, M. 1991a. Bilattices and the semantics of logic programming. *Journal of Logic Programming* 11, 2, 91–116.
- FITTING, M. 1991b. Kleene’s logic, generalized. *Journal of Logic and Computation* 1, 6, 797–810.
- FITTING, M. 1993. The family of stable models. *Journal of Logic Programming* 17, 197–225.
- FITTING, M. 2002. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science* 278, 25–51.
- FITTING, M. 2006. Bilattices are nice things. In *Self-Reference*, T. Bolander, V. Hendricks, and S. A. Pedersen, Eds. CSLI, Stanford, CA, 53–77.
- GINSBERG, M. 1988. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence* 4, 3, 265–316.
- GRIES, D. AND SCHNEIDER, F. B. 1995. Avoiding the undefined by underspecification.

- In *Computer Science Today: Recent Trends and Developments*, J. van Leeuwen, Ed. Lecture Notes in Computer Science, vol. 1000. Springer, 366–373.
- HOGGER, C. 1981. Derivation of logic programs. *Journal of the ACM* 28, 2, 372–392.
- JONES, C. B. AND MIDDELBURG, C. A. 1994. A typed logic of partial functions reconstructed classically. *Acta Informatica* 31, 399–430.
- KLEENE, S. C. 1938. On notation for ordinal numbers. *The Journal of Symbolic Logic* 3, 150–155.
- KOWALSKI, R. A. 1980. *Logic for Problem Solving*. North Holland, New York.
- KOWALSKI, R. A. 1985. The relation between logic programming and logic specification. In *Mathematical Logic and Programming Languages*, C. Hoare and J. Shepherdson, Eds. Prentice-Hall, 11–27.
- KUNEN, K. 1987. Negation in logic programming. *Journal of Logic Programming* 4, 4, 289–308.
- LLOYD, J. W. 1984. *Foundations of Logic Programming*. Springer.
- LOYER, Y., SPYRATOS, N., AND STAMATE, D. 2004. Hypothesis-based semantics of logic programs in multivalued logics. *ACM Transactions on Computational Logic* 5, 3, 508–527.
- MARRIOTT, K. AND SØNDERGAARD, H. 1992. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming* 13, 2&3, 181–204.
- MCCARTHY, J. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Bradford and D. Hirschberg, Eds. North-Holland, 33–70.
- MORRIS, J. M. AND BUNKENBURG, A. 1998. Partiality and nondeterminacy in program proofs. *Formal Aspects of Computing* 10, 76–96.
- MYCROFT, A. 1984. Logic programs and many-valued logic. In *Symposium on Theoretical Aspects of Computer Science*, M. Fontet and K. Mehlhorn, Eds. Lecture Notes in Computer Science, vol. 166. Springer, 274–286.
- NAISH, L. 1996. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, M. Maher, Ed. MIT Press, 185–199.
- NAISH, L. 2000. A three-valued declarative debugging scheme. *Australian Computer Science Communications* 22, 1 (Jan.), 166–173.
- NAISH, L. 2006. A three-valued semantics for logic programmers. *Theory and Practice of Logic Programming* 6, 5, 509–538.
- NAISH, L., SØNDERGAARD, H., AND HORSFALL, B. 2012. Logic programming: From underspecification to undefinedness. In *Theory of Computing 2012*, J. Mestre, Ed. Conferences in Research and Practice in Information Technology, vol. 128. 49–58.
- NISHIMURA, S. 2010. Refining exceptions in four-valued logic. In *Logic-Based Program Synthesis and Transformation*, D. De Schreye, Ed. Lecture Notes in Computer Science, vol. 6037. Springer, 113–127.
- PALMER, D. F. 1997. A parallel implementation of the Andorra Kernel Language. Ph.D. thesis, The University of Melbourne, Australia.
- PEREIRA, L. M. 1986. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, E. Shapiro, Ed. Lecture Notes in Computer Science, vol. 225. Springer, 203–210.
- SATO, T. AND TAMAKI, H. 1984. Transformational logic program synthesis. In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*. 195–201.
- SHAPIRO, E. Y. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge MA.

- SOMOGYI, Z., HENDERSON, F. J., AND CONWAY, T. 1995. Mercury: An efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*. Glenelg, Australia, 499–512.
- TARSKI, A. 1955. A lattice-theoretical theorem and its applications. *Pacific Journal of Mathematics* 5, 285–309.
- VAN EMDEN, M. AND KOWALSKI, R. 1976. The semantics of logic as a programming language. *Journal of the ACM* 23, 733–742.
- VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38, 3, 620–650.